JCodeFlow with Multi-thread Support

This article highlights further applications of the JCodeFlow tool we discussed in LFY last month, to cover multi-threaded programs. Read on to get enlightened...

In the August 2005 issue of LFY we introduced you to the JCodeFlow tool that can be used to trace programs written in Java. But one of the most important features missing was the support to get traces from multi-threaded Java programs. In this article we will show you how one could extend the JCodeFlow tool so that it can trace even the programs that spawn multiple threads.

A thread (also called an *execution context* or *lightweight process*) is a single sequential flow of control within a program. Multi-threaded programs usually spawn one or more threads so that more than one activity can be performed in parallel, within the same application. When the JCodeFlow tool (without multi-thread support) is used to trace such programs, the report created will be of no use as the report will be interleaved with trace information from different threads spawned by the traced program, as shown in

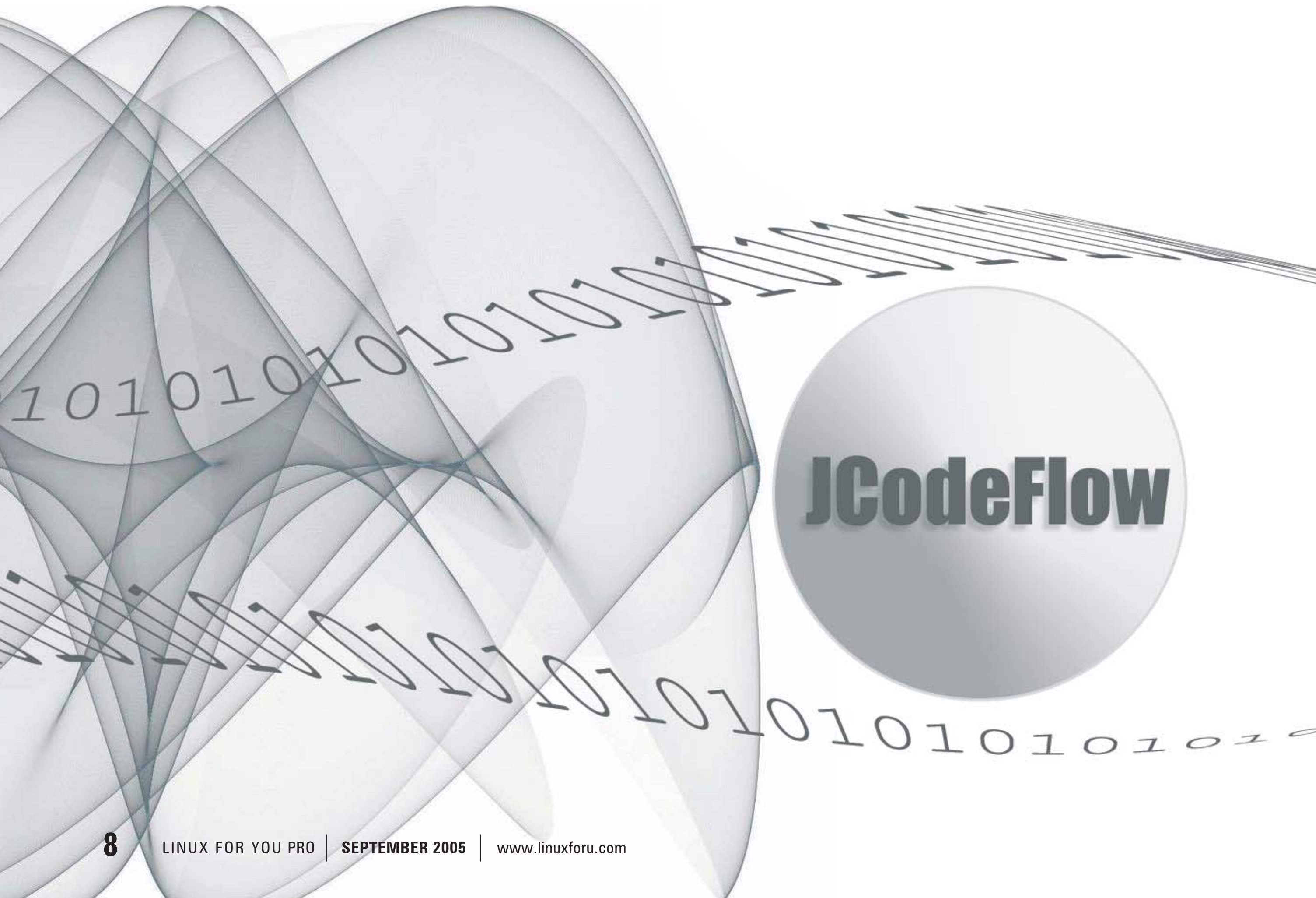

Figure 1: Incorrect JCodeFlow output for multi-threaded program

```
main(52, MultiThreaded.java, main)
    <init>(32, MultiThreaded.java, main)
    go(35, MultiThreaded.java, main)
        <init>(3, MultiThreaded.java, main)
        <init>(3, MultiThreaded.java, main)
        run(14, MultiThreaded.java, ThreadA)
            run(14, MultiThreaded.java, ThreadB)
                display1(6, MultiThreaded.java, ThreadA)
                display1(6, MultiThreaded.java, ThreadB)
                display2(10, MultiThreaded.java, ThreadA)
                    display2(10, MultiThreaded.java, ThreadB)
                display1(6, MultiThreaded.java, ThreadB)
                    display1(6, MultiThreaded.java, ThreadA)
                display2(10, MultiThreaded.java, ThreadA)
                    display2(10, MultiThreaded.java, ThreadB)
                display1(6, MultiThreaded.java, ThreadA)
                display1(6, MultiThreaded.java, ThreadB)
                display2(10, MultiThreaded.java, ThreadB)
                    display2(10, MultiThreaded.java, ThreadA)
                display1(6, MultiThreaded.java, ThreadA)
                display1(6, MultiThreaded.java, ThreadB)
                display2(10, MultiThreaded.java, ThreadA)
                display2(10, MultiThreaded.java, ThreadB)
                display1(6, MultiThreaded.java, ThreadB)
                    display1(6, MultiThreaded.java, ThreadA)
                display2(10, MultiThreaded.java, ThreadA)
            display2(10, MultiThreaded.java, ThreadB)
```

Figure 1. To overcome this, we need to modify our JCodeFlow tool so that it can report trace information for each thread separately.

Here is a quick recap on what the JCodeFlow tool is. JCodeFlow is a simple, light-weight Java Platform Debugger Architecture (JPDA)-based tool that can be used to trace any programs written in Java. It uses the JPDA *MethodEntryEvent* and *MethodExitEvent* to get the code flow (trace) data. By using the JCodeFlow tool, you can easily understand the flow of a program in no time. The JCodeFlow tool has been designed in such a way that it can connect to the remote Java Virtual Machine (JVM) running a Java application.

Listing 1 shows a sample multi-threaded program that we will trace using our modified JCodeFlow tool. Let's name the sample program as MultiThreaded.java

Listing 1: A sample multi-threaded program

```
class WorkerThread extends Thread
{
    public WorkerThread(String threadName) {
setName(threadName); }

    private void display1(int val) {
        System.out.println(getName() + ": " + val);
    }

    private void display2(int val) {
        System.out.println(getName() + ": " + val);
    }

    public void run() {
        int i = 0;

        for( i=0; i<10; i++ ) {
            if( (i % 2) == 0 ) {
                display1(i);
            } else {
                display2(i);
            }

            try {
                Thread.sleep(10);
            } catch(Throwable ignore) {
            }
        }
    }
}


public class MultiThreaded
{
    public void go() throws Throwable {
        WorkerThread wt1 = null, wt2 = null;

        try {
            wt1 = new WorkerThread("ThreadA");
            wt2 = new WorkerThread("ThreadB");

            wt1.start();
            wt2.start();

            wt1.join();
            wt2.join();
        } catch(Throwable t) {
            throw t;
        }
    }

    public static void main(String args[]) {
        MultiThreaded mt = null;

        try {
            mt = new MultiThreaded();
            mt.go();
        } catch(Throwable t) {
            t.printStackTrace();
        }
    }
}
```

MultiThreaded.java is a simple program that spawns two threads: Thread A and Thread B. The threads' run method alternatively calls two functions: *display1* and *display2,* to display an integer value.

Let's modify our JCodeFlow tool to trace multi-thread Java programs like our sample *MultiThreaded.java*. In order to print the method details separately for each thread, we will define a new class to hold thread details as shown in Listing 2.

```
Listing 2

class ThreadDetails
{
    private long indent = 0;
    private PrintWriter pw = null;

    public ThreadDetails(long indent, PrintWriter pw) {
        this.indent = indent;
        this.pw = pw;
    }

    public void setIndent(long indent) { this.indent = indent;
}
    public long getIndent() { return this.indent; }
    public PrintWriter getWriter() { return this.pw; }
    public String toString() { return "indent: " + indent + ",
pw: " + pw; }
}
```

We also need to modify the method exit event handler function *handleMethodExitEvent* to get the indent value from the *ThreadDetails* class for the thread, as shown in Listing 3.

```
Listing 3

    private void handleMethodExitEvent(MethodExitEvent event)
throws Throwable {
        long indent = 0;

        try {
            Method method = event.method();

            if( method.isSynthetic() || method.isAbstract()  ||
method.isNative() ) {
                return;
            }

            threadDetails = (ThreadDetails)
threadMap.get(event.thread().name());
            indent = threadDetails.getIndent();
            indent—;

            threadDetails.setIndent(indent);
            threadMap.put(event.thread().name(),threadDetails);
        } catch(Throwable t) {
            throw t;
        }
    }
```

Similarly, we also need to modify the *handleMethodEntryEvent* function to read and update thread details from the *ThreadDetails* class, as shown in Listing 4.

```
Listing 4

    private void handleMethodEntryEvent(MethodEntryEvent event)
throws Throwable {
        PrintWriter pw = null;
```

```
        try {
            Method method = event.method();

            if( method.isSynthetic() || method.isAbstract()  ||
method.isNative() ) {
                return;
            }

           printMethodDetails(event.thread().name(),method.name()
+
                "(" + (method.location().lineNumber()) + ", " +
                method.location().sourceName() + ")");
        } catch(Throwable t) {
            throw t;
        }
    }

    private void printMethodDetails(String threadName, String
methodDetails)
        throws Throwable
    {
        long indent = 0;
        PrintWriter pw = null;

        try {
            if( !threadMap.containsKey(threadName) ) {
                pw = new PrintWriter(new
FileOutputStream(threadName + ".txt"),true);
                threadDetails = new ThreadDetails(indent,pw);
                threadMap.put(threadName,threadDetails);
            }

            threadDetails = (ThreadDetails)
threadMap.get(threadName);
            pw = threadDetails.getWriter();
            indent = threadDetails.getIndent();

            for( long i=0; i<indent; i++ ) { pw.print("\t"); }
            pw.println(methodDetails);

            indent++;

            threadDetails.setIndent(indent);
            threadMap.put(threadName,threadDetails);
        } catch(Throwable t) {
            throw t;
        }
    }
```

That's it. Now we are ready to compile and run our modified JCodeFlow tool.

- Copy the *JCodeFlow.java* and *MultiThreaded.java* files from the LFY CD-ROM to your system hard disk and compile them.
- First run the *MultiThreaded.java* program as shown below:

```
java -
Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=4000 -
Xdebug MultiThreaded
```

- Next start the JCodeFlow tool as shown:

```
java -cp tools.jar;. JcodeFlow
```

The JCodeFlow tool will create <threadname>.txt files for each of the Java threads. For our *MultiThreaded.java* it will create three files. Their content is shown below:

```
main.txt

main(52, MultiThreaded.java)
        <init>(32, MultiThreaded.java)
        go(35, MultiThreaded.java)
                 <init>(3, MultiThreaded.java)
                 <init>(3, MultiThreaded.java)
```

```
ThreadA.txt

run(14, MultiThreaded.java)
        display1(6, MultiThreaded.java)
        display2(10, MultiThreaded.java)
        display1(6, MultiThreaded.java)
        display2(10, MultiThreaded.java)
        display1(6, MultiThreaded.java)
        display2(10, MultiThreaded.java)
        display1(6, MultiThreaded.java)
        display2(10, MultiThreaded.java)
        display1(6, MultiThreaded.java)
        display2(10, MultiThreaded.java)
```

```
ThreadB.txt

run(14, MultiThreaded.java)
        display1(6, MultiThreaded.java)
        display2(10, MultiThreaded.java)
        display1(6, MultiThreaded.java)
        display2(10, MultiThreaded.java)
        display1(6, MultiThreaded.java)
        display2(10, MultiThreaded.java)
        display1(6, MultiThreaded.java)
        display2(10, MultiThreaded.java)
        display1(6, MultiThreaded.java)
        display2(10, MultiThreaded.java)
```

From the above output we could see how easy it is to understand even the multi-threaded programs written in Java, using our modified JCodeFlow code trace tool. Happy debugging!

Acknowledgements

- **JDI**—http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jdi/index.html?overview-summary.html
- **JPDA**—http://java.sun.com/products/jpda/

LFY

By: Raja R.K. The author is a lead engineer with HCL Technologies, Chennai.